python for Computational Problem Solving - pCPS - Data and Expressions Lecture Slides - Class #7 to Class#8

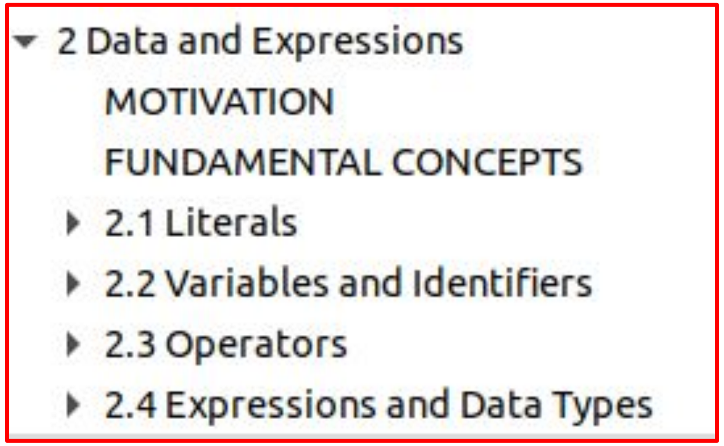
Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

python for Computational Problem Solving Syllabus

Unit I: Computational Problem Solving - 12 Hours

Limits of Computational Problem Solving - Computer Algorithm - Computer Hardware - Digital Computer - Operating System- Limits of IC technology - Computer Software - Syntax, semantics and program translation ,Introduction to Python Programming Language, IDLE Python Development Environment, Output function - variables, types and id,input function , operators and expressions, Control structures .

T1: 1.1 – 1.7
T1: 2.1 - 2.4
T1: 3.1 – 3.4



- 2 Data and Expressions
    - MOTIVATION
    - FUNDAMENTAL CONCEPTS
    - 2.1 Literals
    - 2.2 Variables and Identifiers
    - 2.3 Operators
    - 2.4 Expressions and Data Types

pCPS 2.2.2 Variables and Keyboard Input

- The **value** that is assigned to a given **variable** **need** **not** have to be specified in the program

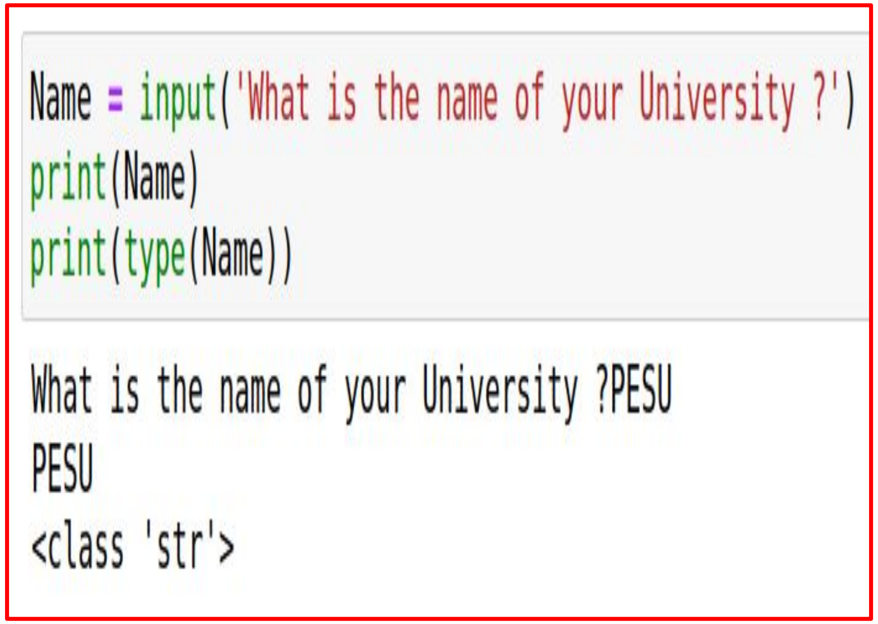
- The **value** can come from the user by use of the input function

- As shown on the side, the variable name is assigned the string **'PESU'**.

- If the **user** **hit** **return** without entering any value, name would be assigned to the **empty** **string** ('')

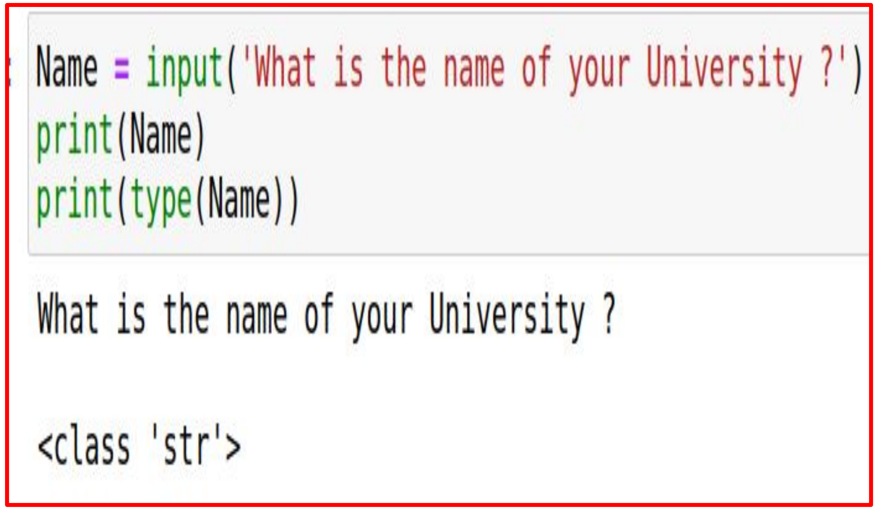
- All input is returned by the input function as a string type.

- For the input of numeric values, the response must be converted to the appropriate type.



```
Name = input('What is the name of your University ?')
print(Name)
print(type(Name))
```

```
What is the name of your University ?PESU
PESU
<class 'str'>
```



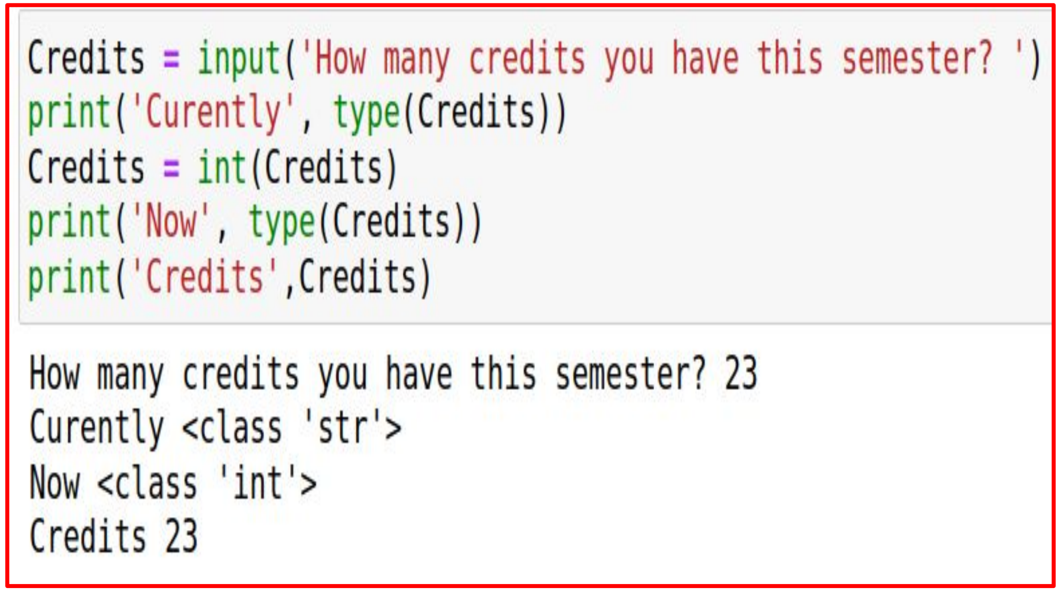
```
Name = input('What is the name of your University ?')
print(Name)
print(type(Name))
```

```
What is the name of your University ?

<class 'str'>
```

pCPS 2.2.2 Variables and Keyboard Input

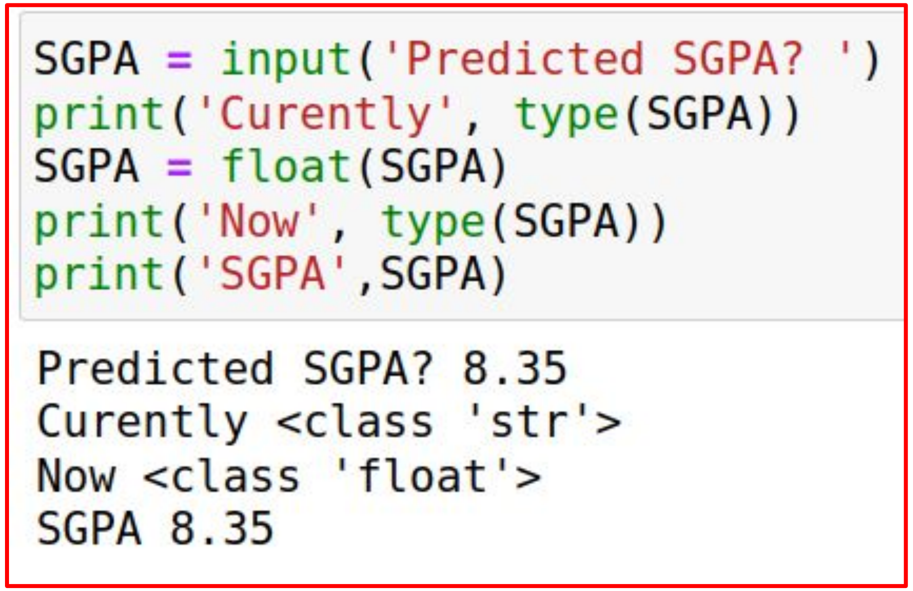
- For the input of numeric values, the response must be converted to the appropriate type.

- python provides built-in type conversion functions int () and float () for this purpose



```python
Credits = input('How many credits you have this semester? ')
print('Curently', type(Credits))
Credits = int(Credits)
print('Now', type(Credits))
print('Credits',Credits)
```

```
How many credits you have this semester? 23
Curently <class 'str'>
Now <class 'int'>
Credits 23
```

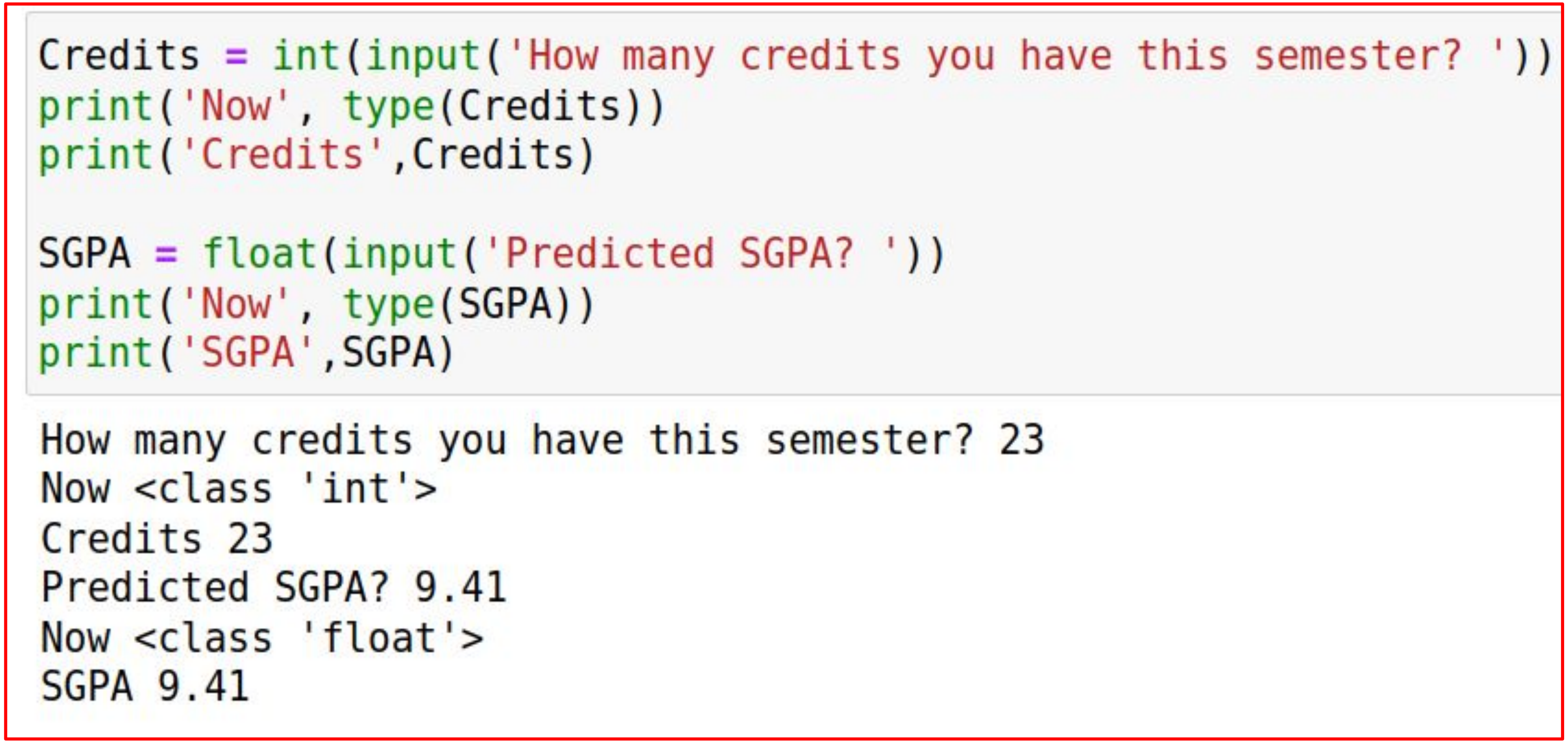


```python
SGPA = input('Predicted SGPA? ')
print('Curently', type(SGPA))
SGPA = float(SGPA)
print('Now', type(SGPA))
print('SGPA',SGPA)
```

```
Predicted SGPA? 8.35
Curently <class 'str'>
Now <class 'float'>
SGPA 8.35
```

pCPS 2.2.2 Variables and Keyboard Input

- Note that the program lines above could be combined as follows



```python
Credits = int(input('How many credits you have this semester? '))
print('Now', type(Credits))
print('Credits',Credits)

SGPA = float(input('Predicted SGPA? '))
print('Now', type(SGPA))
print('SGPA',SGPA)
```

```
How many credits you have this semester? 23
Now <class 'int'>
Credits 23
Predicted SGPA? 9.41
Now <class 'float'>
SGPA 9.41
```

pCPS 2.2.3 Identifiers in python

- An **identifier** is a sequence of one or more characters used to provide a name for a given program element.

- Variable names line, Credits, and SGPA are each identifiers.

- python is **Case Sensitive** , thus, Credits is different from credits

- **Identifiers** may contain **letters** and **digits**, but **cannot** **begin** with a **digit**.

- The underscore character, **_**, is also allowed to aid in the readability of long identifier names.

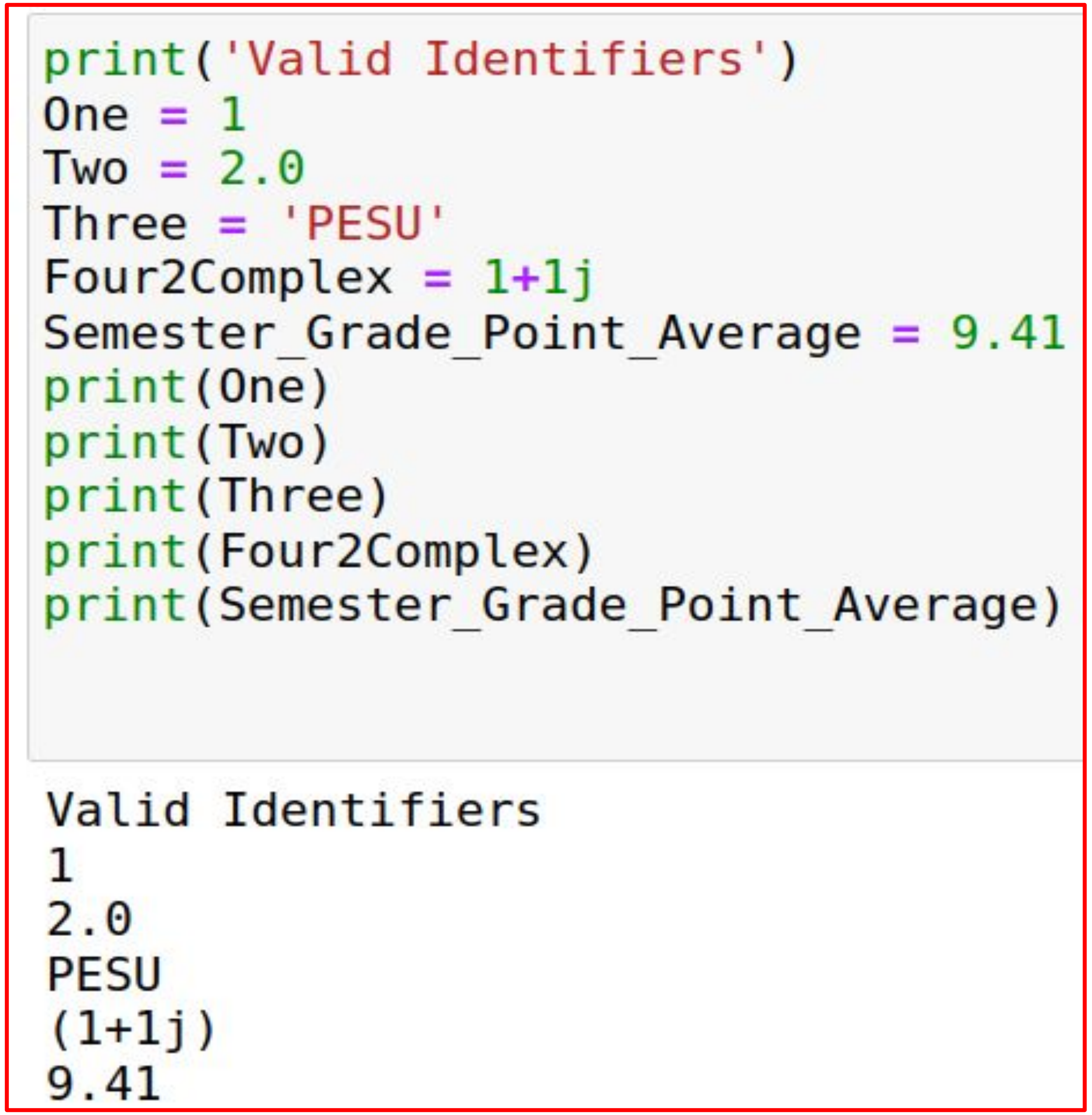
- **_ (underscore)** should not be used as the first character, however, as identifiers beginning with an underscore have special meaning in python

- Spaces are not allowed as part of an **identifier**.

- This is a common error since some operating systems allow spaces within file names.

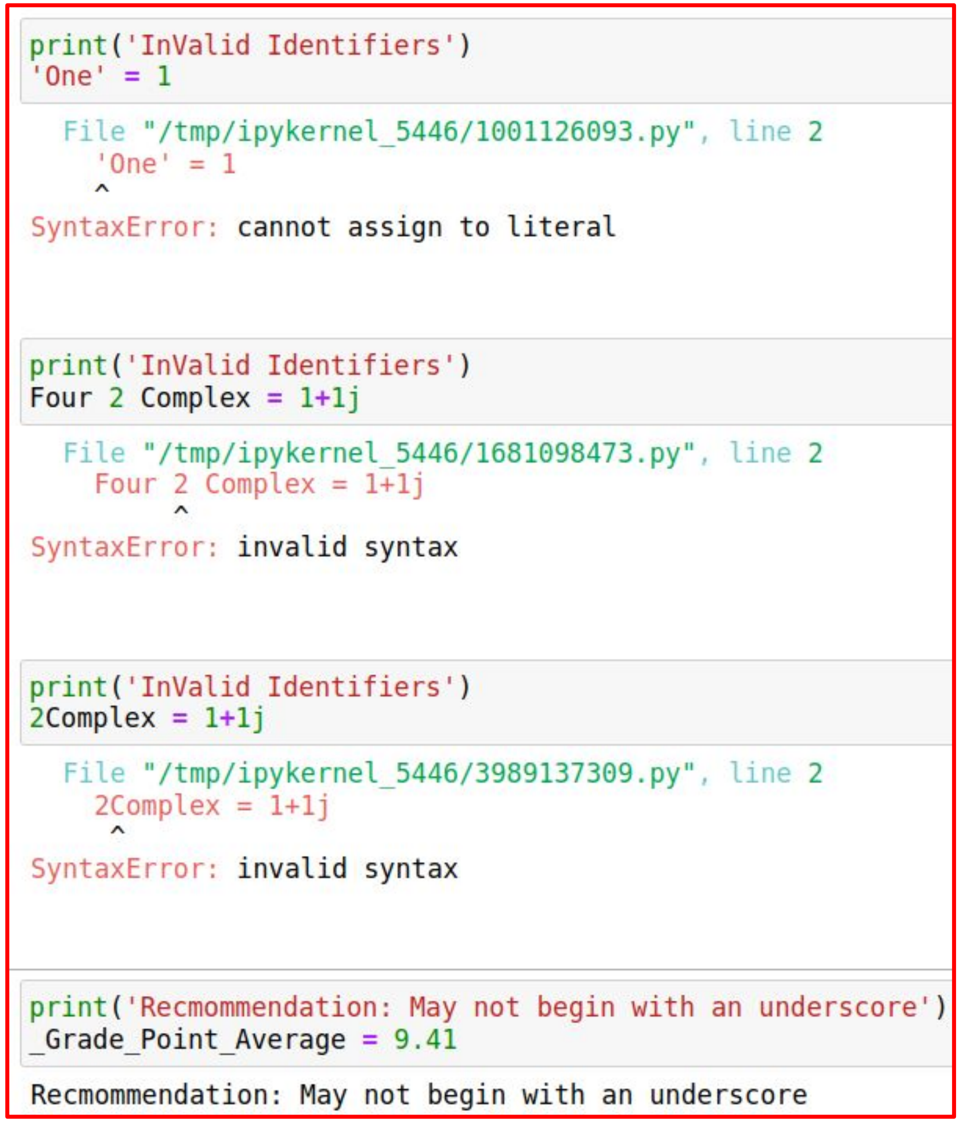
- Any **identifier** containing a space character would be considered two separate **identifiers**

pCPS 2.2.3 Identifiers in python



```
print('Valid Identifiers')
One = 1
Two = 2.0
Three = 'PESU'
Four2Complex = 1+1j
Semester_Grade_Point_Average = 9.41
print(One)
print(Two)
print(Three)
print(Four2Complex)
print(Semester_Grade_Point_Average)
```

```
Valid Identifiers
1
2.0
PESU
(1+1j)
9.41
```



```
print('InValid Identifiers')
'One' = 1
```

```
  File "/tmp/ipykernel_5446/1001126093.py", line 2
    'One' = 1
    ^
SyntaxError: cannot assign to literal
```

```
print('InValid Identifiers')
Four 2 Complex = 1+1j
```

```
  File "/tmp/ipykernel_5446/1681098473.py", line 2
    Four 2 Complex = 1+1j
         ^
SyntaxError: invalid syntax
```

```
print('InValid Identifiers')
2Complex = 1+1j
```

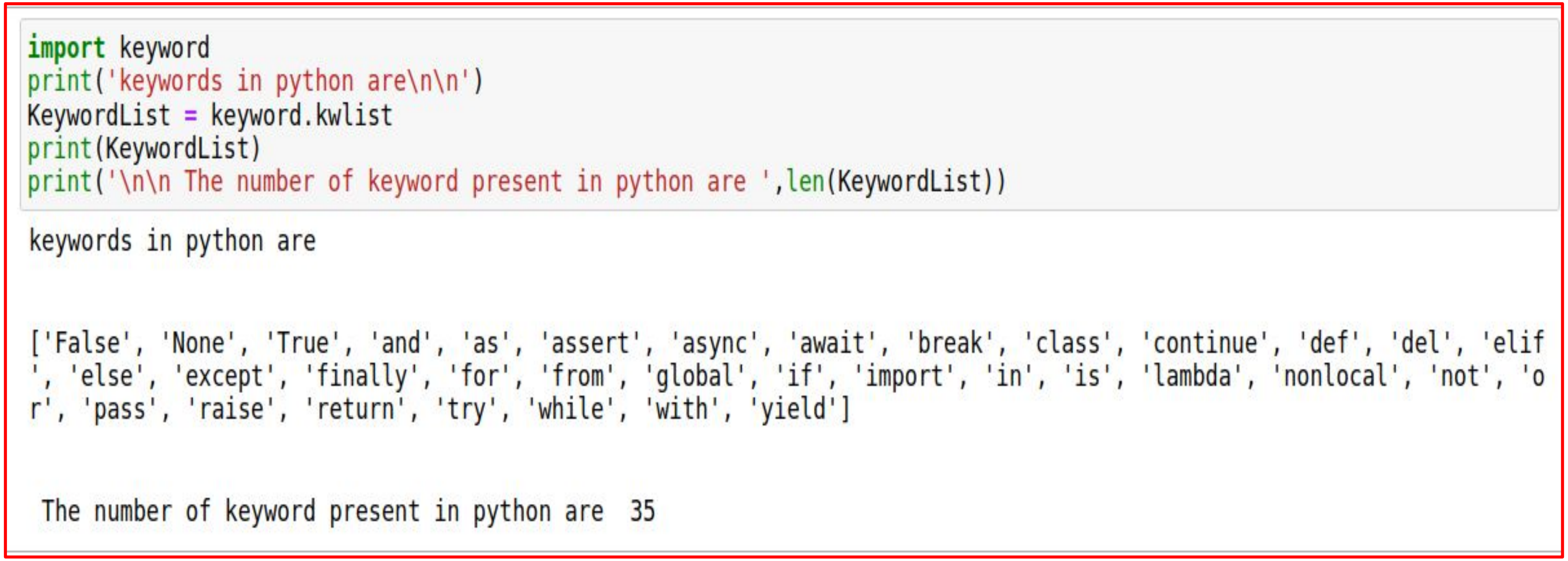
```
  File "/tmp/ipykernel_5446/3989137309.py", line 2
    2Complex = 1+1j
     ^
SyntaxError: invalid syntax
```

```
print('Recmommendation: May not begin with an underscore')
_Grade_Point_Average = 9.41
```

```
Recmommendation: May not begin with an underscore
```

pCPS 2.2.4 keywords and Other Predefined Identifiers in python

- A **keyword** is an identifier that has **predefined** meaning in a programming language.
- keywords **cannot** be used as “**regular**” identifiers.
- There are 35 keywords in python

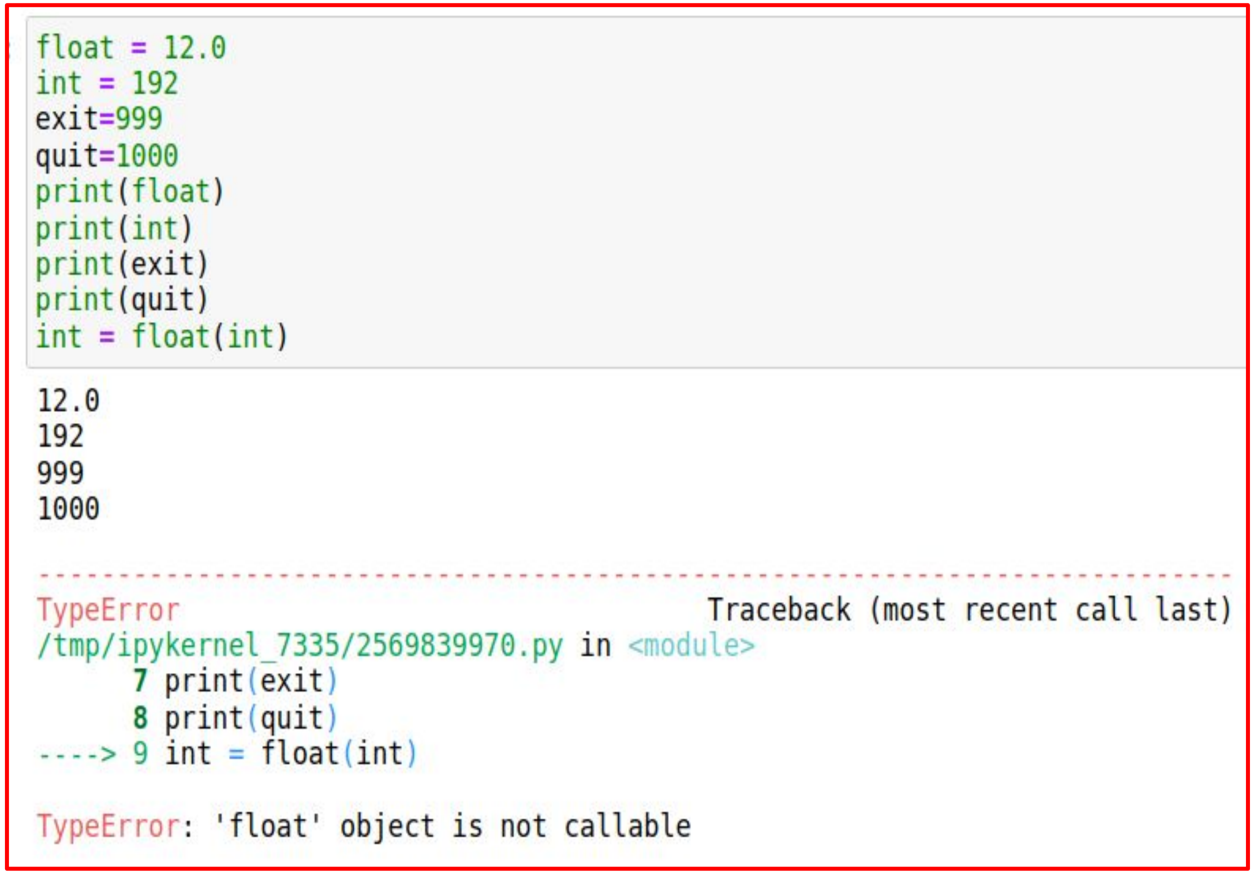


```python
import keyword
print('keywords in python are\n\n')
KeywordList = keyword.kwlist
print(KeywordList)
print('\n\n The number of keyword present in python are ',len(KeywordList))
```

```
keywords in python are


['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif
', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'o
r', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']


 The number of keyword present in python are  35
```

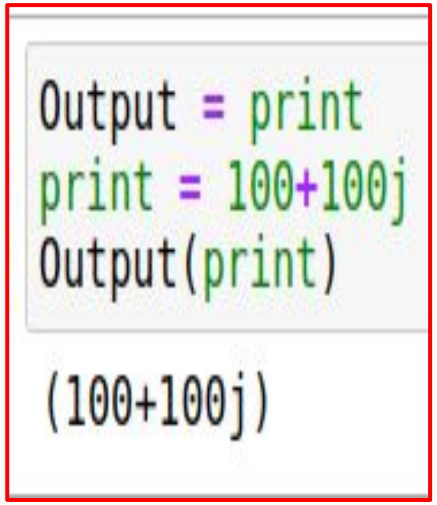
pCPS 2.2.4 keywords and Other Predefined Identifiers in python

- There are other predefined identifiers that can be used as regular identifiers, but should not be.
- This includes float, int, print, exit, and quit



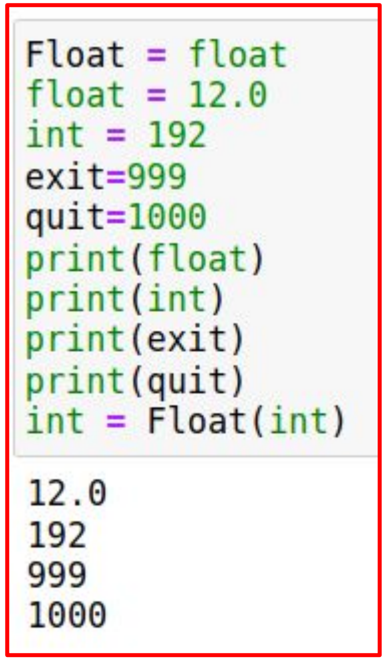
```
float = 12.0
int = 192
exit=999
quit=1000
print(float)
print(int)
print(exit)
print(quit)
int = float(int)
```

```
12.0
192
999
1000

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_7335/2569839970.py in <module>
      7 print(exit)
      8 print(quit)
----> 9 int = float(int)

TypeError: 'float' object is not callable
```



```
Output = print
print = 100+100j
Output(print)
```

```
(100+100j)
```



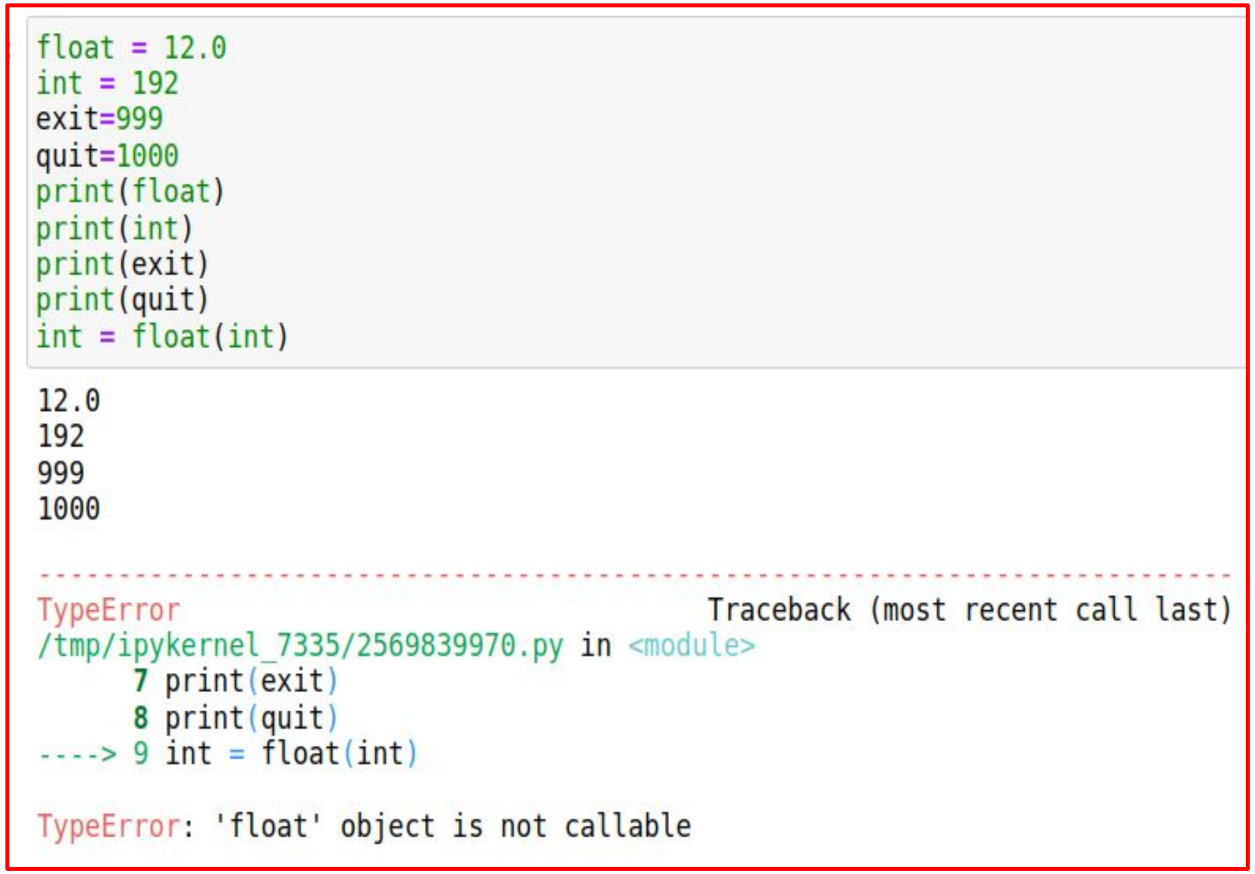
```
Float = float
float = 12.0
int = 192
exit=999
quit=1000
print(float)
print(int)
print(exit)
print(quit)
int = Float(int)
```

```
12.0
192
999
1000
```

pCPS 2.2.4 keywords and Other Predefined Identifiers in python

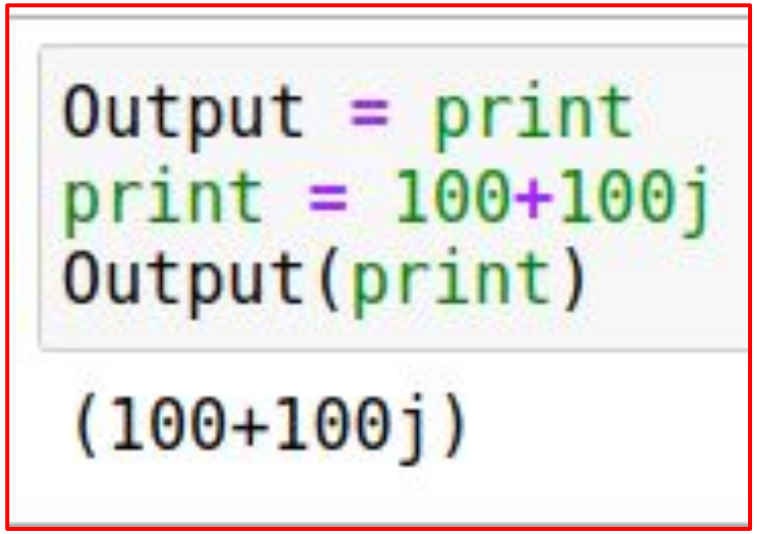
- There are other predefined identifiers that can be used as regular identifiers, but should not be.

- This includes float, int, print, exit, and quit



```
float = 12.0
int = 192
exit=999
quit=1000
print(float)
print(int)
print(exit)
print(quit)
int = float(int)
```

```
12.0
192
999
1000

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_7335/2569839970.py in <module>
      7 print(exit)
      8 print(quit)
----> 9 int = float(int)

TypeError: 'float' object is not callable
```



```
Output = print
print = 100+100j
Output(print)
```

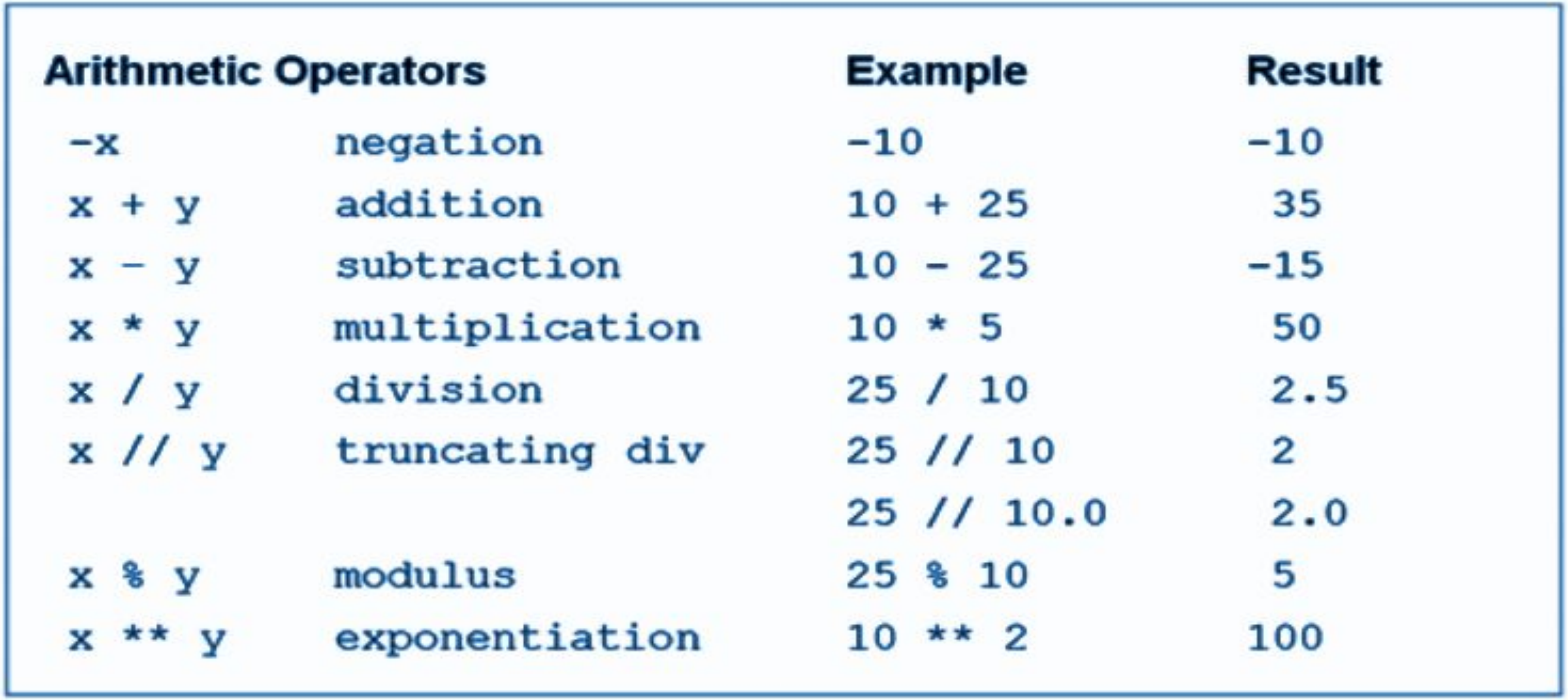
```
(100+100j)
```

pCPS 2.3.1 Operator

- An **operator** is a symbol that represents an operation that may be performed on one or more operands .
- An **operand** is a value that a given operator is applied to
- A **unary** operator operates on only one operand
- A **binary** operator operates on two operands
- **Most** operators in **programming languages** are **binary operators**.

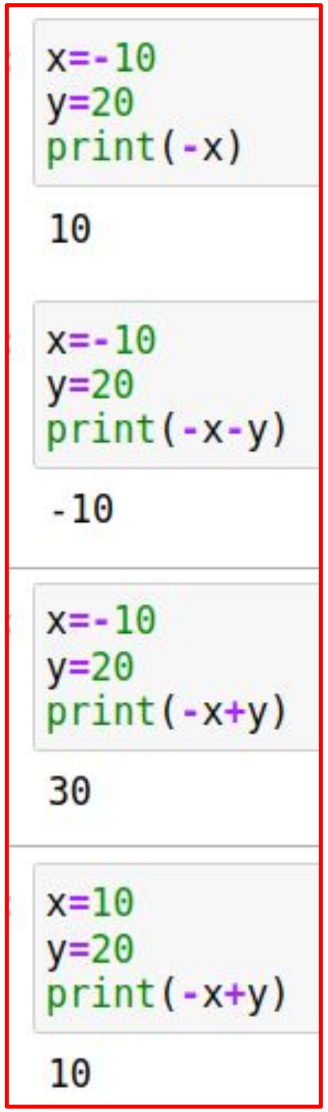
pCPS 2.3.2 Arithmatic Operators

- python provides the arithmetic operators as given below



Arithmetic Operators		Example	Result
-x	negation	-10	-10
x + y	addition	10 + 25	35
x - y	subtraction	10 - 25	-15
x * y	multiplication	10 * 5	50
x / y	division	25 / 10	2.5
x // y	truncating div	25 // 10	2
		25 // 10.0	2.0
x % y	modulus	25 % 10	5
x ** y	exponentiation	10 ** 2	100

pCPS 2.3.2 Arithmatic Operators



```
x=-10
y=20
print(-x)
```

```
10
```

```
x=-10
y=20
print(-x-y)
```

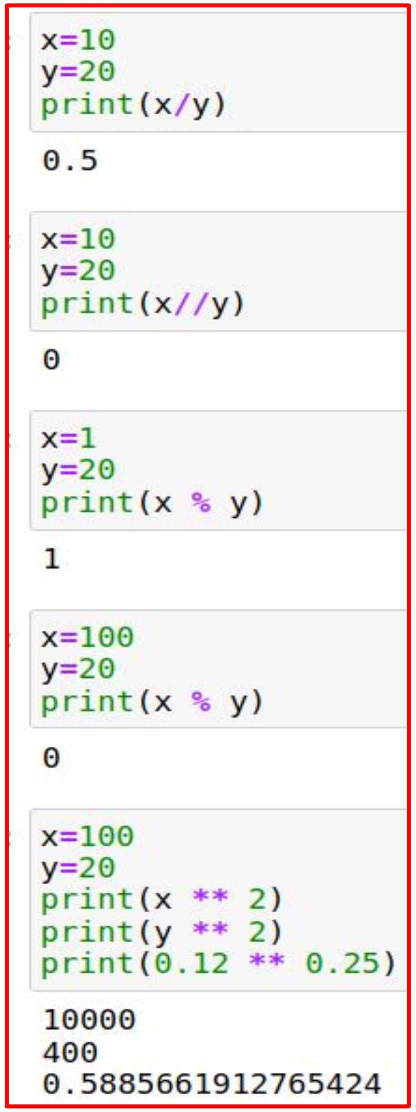
```
-10
```

```
x=-10
y=20
print(-x+y)
```

```
30
```

```
x=10
y=20
print(-x+y)
```

```
10
```



```
x=10
y=20
print(x/y)
```

```
0.5
```

```
x=10
y=20
print(x//y)
```

```
0
```

```
x=1
y=20
print(x % y)
```

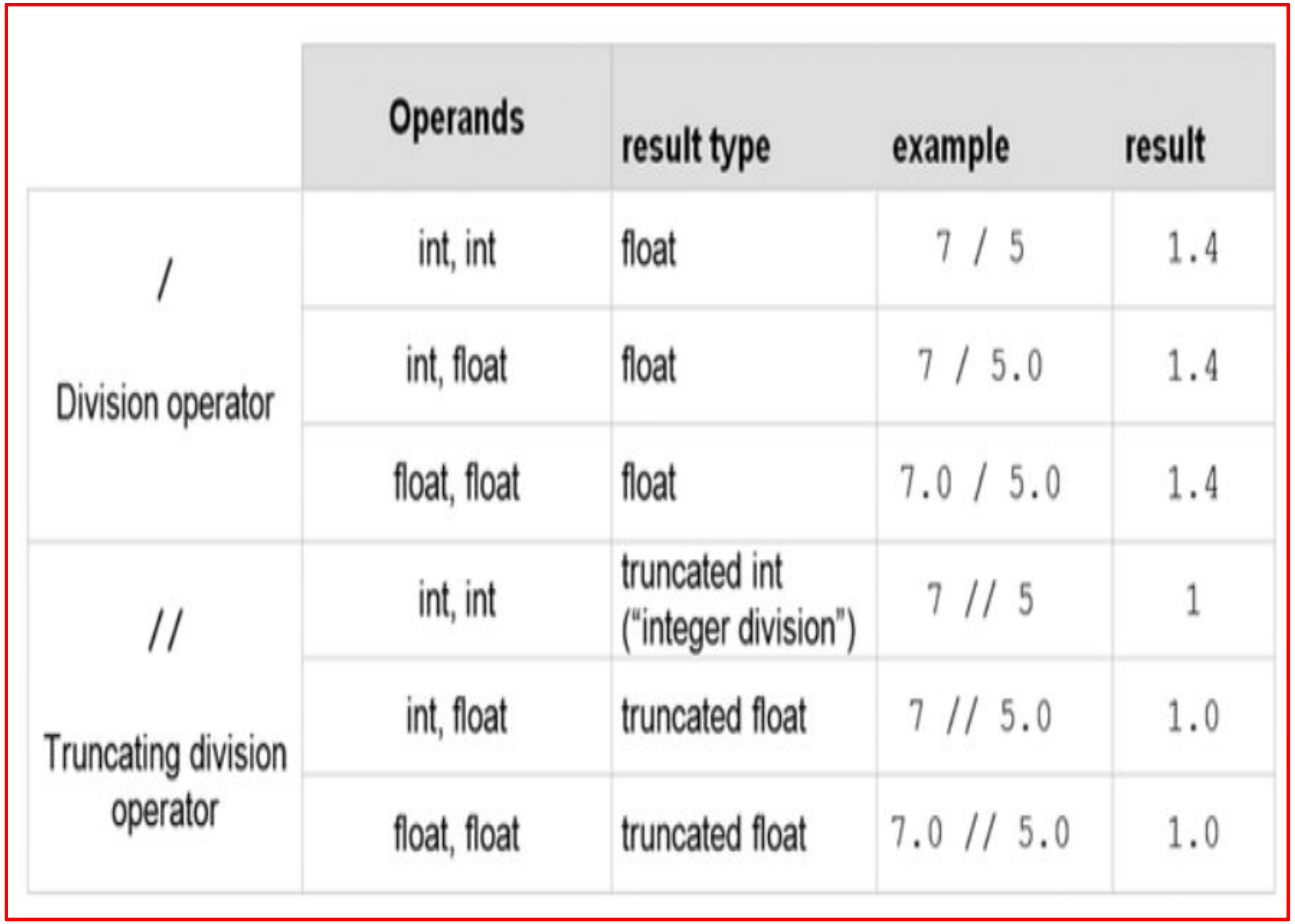
```
1
```

```
x=100
y=20
print(x % y)
```

```
0
```

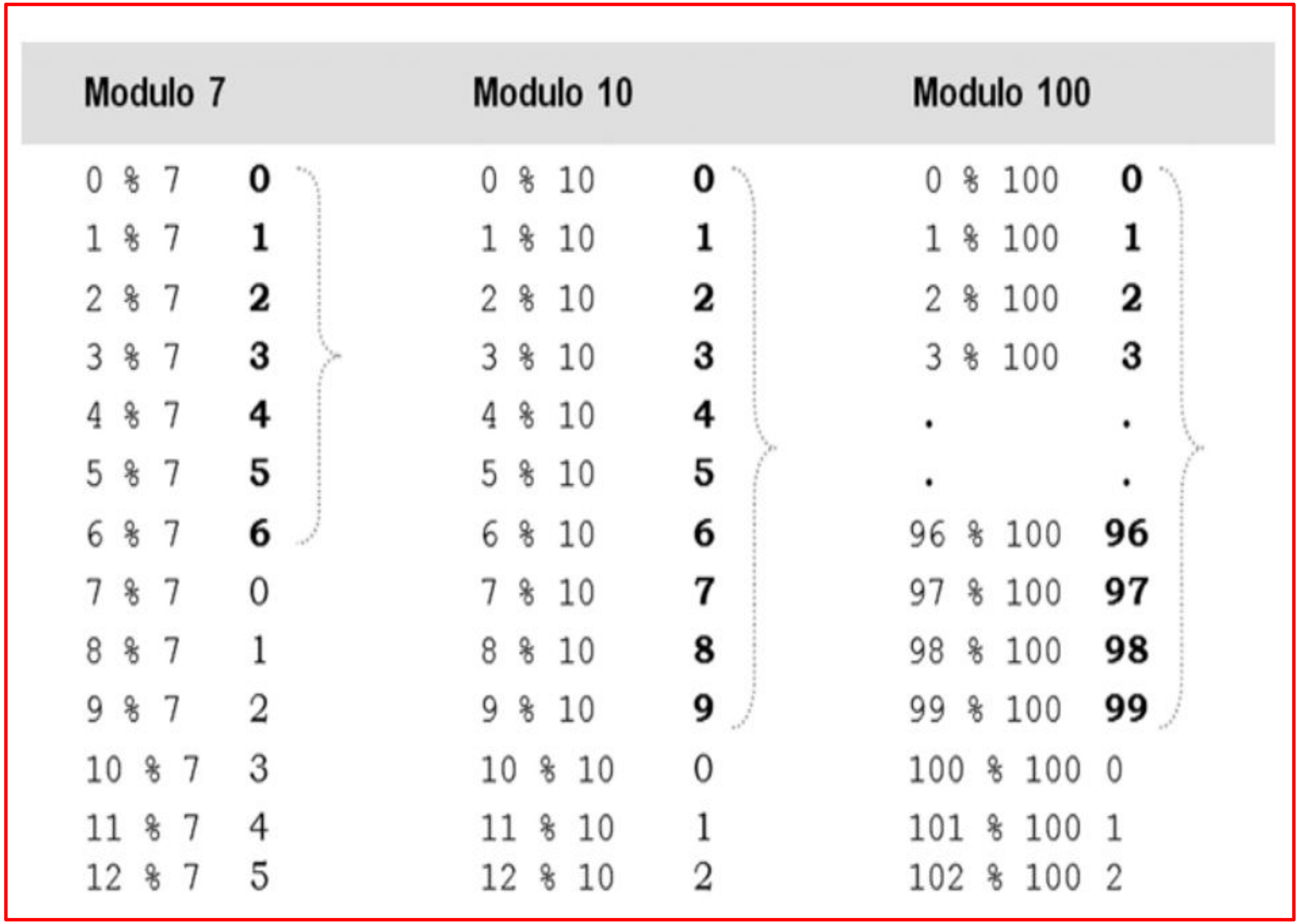
```
x=100
y=20
print(x ** 2)
print(y ** 2)
print(0.12 ** 0.25)
```

```
10000
400
0.5885661912765424
```



	Operands	result type	example	result
/ Division operator	int, int	float	7 / 5	1.4
	int, float	float	7 / 5.0	1.4
	float, float	float	7.0 / 5.0	1.4
// Truncating division operator	int, int	truncated int ("integer division")	7 // 5	1
	int, float	truncated float	7 // 5.0	1.0
	float, float	truncated float	7.0 // 5.0	1.0

pCPS 2.3.2 Arithmatic Operators

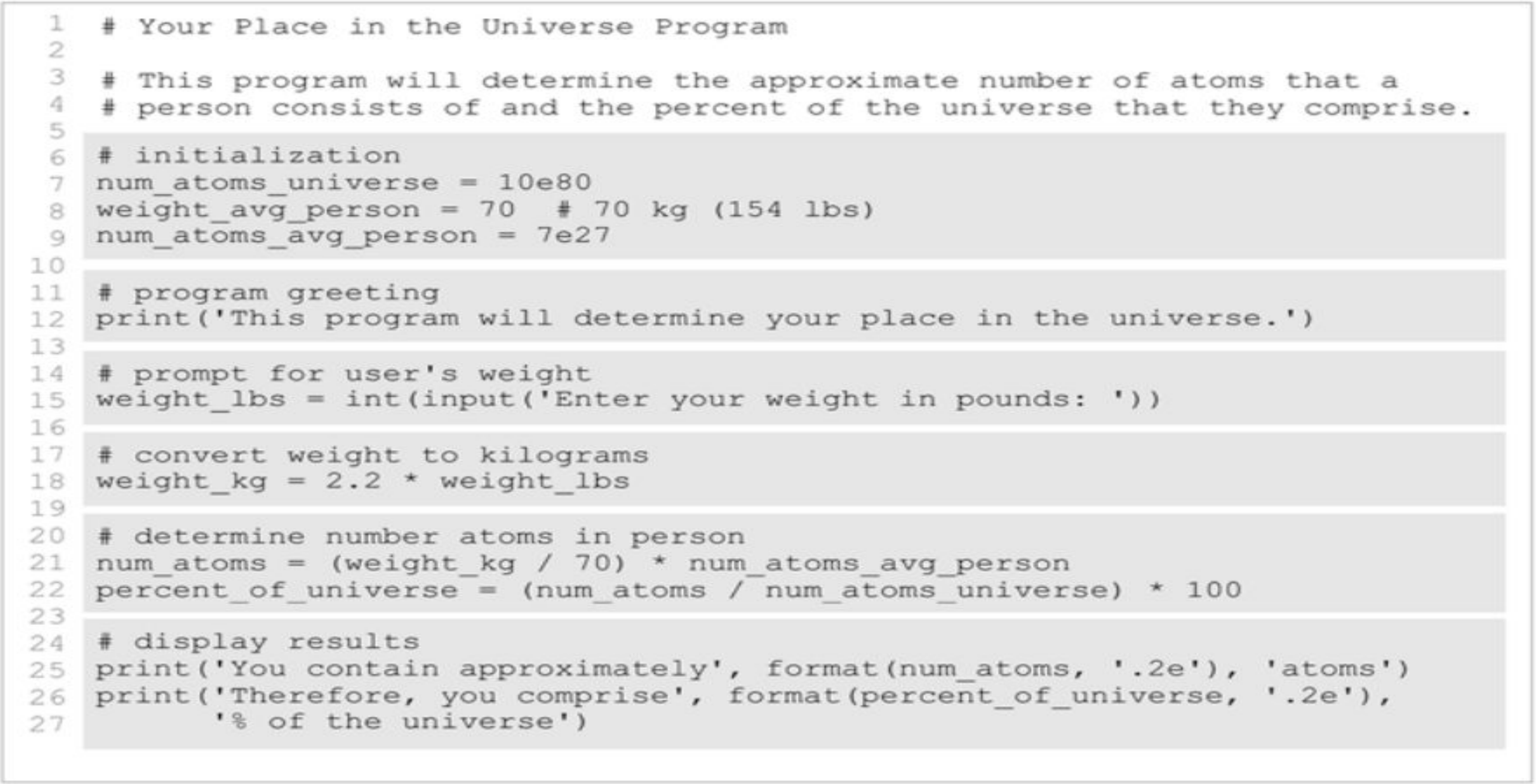


Modulo 7		Modulo 10		Modulo 100	
0 % 7	**0**	0 % 10	**0**	0 % 100	**0**
1 % 7	**1**	1 % 10	**1**	1 % 100	**1**
2 % 7	**2**	2 % 10	**2**	2 % 100	**2**
3 % 7	**3**	3 % 10	**3**	3 % 100	**3**
4 % 7	**4**	4 % 10	**4**	.	.
5 % 7	**5**	5 % 10	**5**	.	.
6 % 7	**6**	6 % 10	**6**	96 % 100	**96**
7 % 7	0	7 % 10	**7**	97 % 100	**97**
8 % 7	1	8 % 10	**8**	98 % 100	**98**
9 % 7	2	9 % 10	**9**	99 % 100	**99**
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

- The **modulus operator (%)** gives the remainder of the division of its operands, resulting in a cycle of values as shown in the figure on the side

- The modulus and runcating (integer) division operators are complements of each other.

pCPS 2.3.3 Let’s Apply it



```
# Your Place in the Universe Program

# This program will determine the approximate number of atoms that a
# person consists of and the percent of the universe that they comprise.

# initialization
num_atoms_universe = 10e80
weight_avg_person = 70  # 70 kg (154 lbs)
num_atoms_avg_person = 7e27

# program greeting
print('This program will determine your place in the universe.')

# prompt for user's weight
weight_lbs = int(input('Enter your weight in pounds: '))

# convert weight to kilograms
weight_kg = 2.2 * weight_lbs

# determine number atoms in person
num_atoms = (weight_kg / 70) * num_atoms_avg_person
percent_of_universe = (num_atoms / num_atoms_universe) * 100

# display results
print('You contain approximately', format(num_atoms, '.2e'), 'atoms')
print('Therefore, you comprise', format(percent_of_universe, '.2e'),
      '% of the universe')
```

pCPS 2.4 Expressions and Data Types

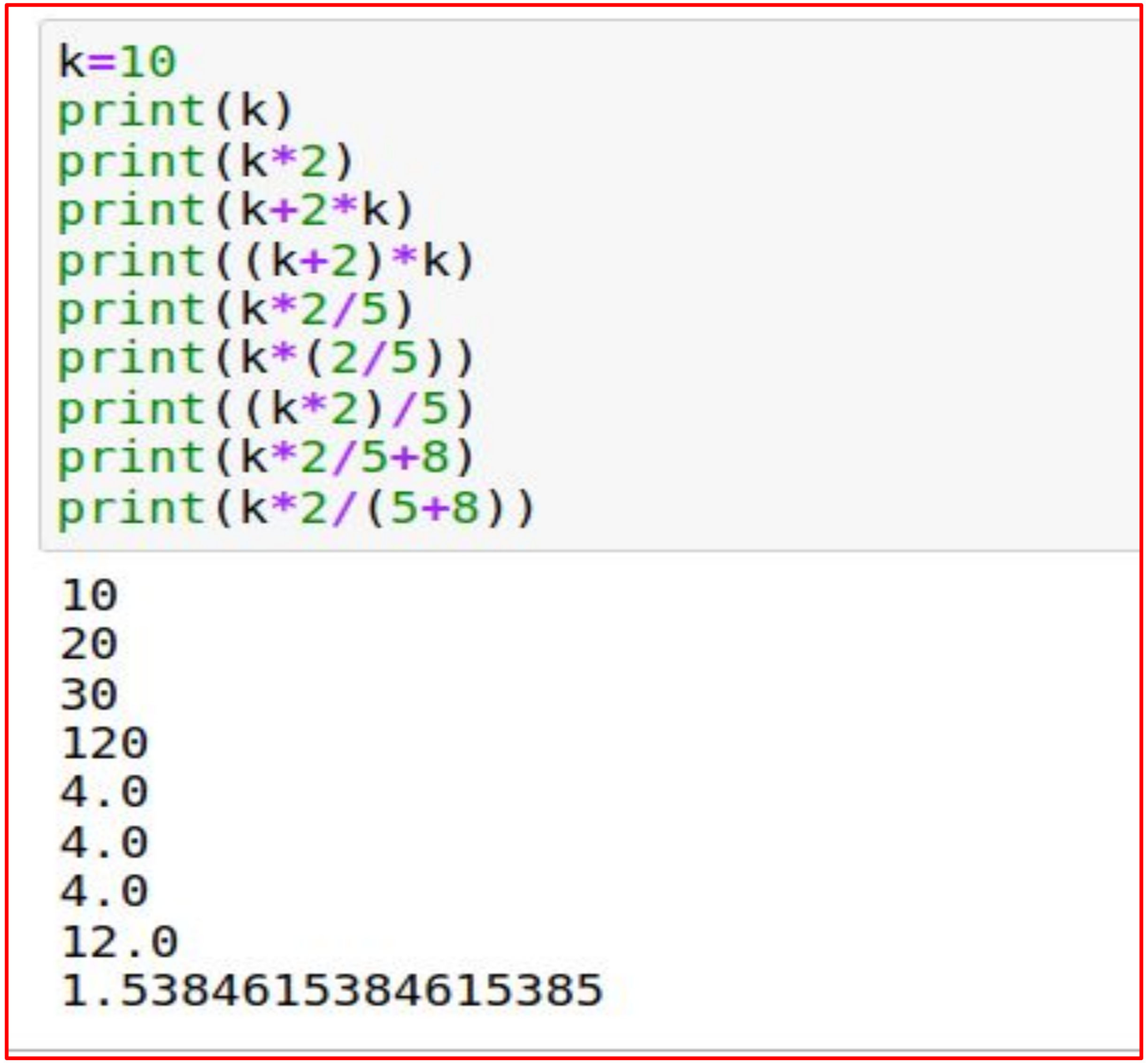
- **Operators** and **Operands** can be **combined** to form **expressions**.

- Process of Arithmetic expressions are evaluation in python.

pCPS 2.4.1 Expressions in python

- An **Expression** is a combination of symbols that evaluates to a value.
- **Expressions**, most commonly, consist of a **combination** of **operators** and **operands**
- An **expression** can also consist of a **single literal** or **variable** or also a **sub expression**
- **Expressions** that **evaluate** to a **numeric** type are called **arithmetic expressions** .
- A **subexpression** is any expression that is **part** of a **larger expression**.
- **Subexpressions** may be denoted by the use of **parentheses**
- If **no parentheses** are used, then an expression is evaluated according to the **rules** of **operator precedence** in python

pCPS 2.4.1 Expressions in python



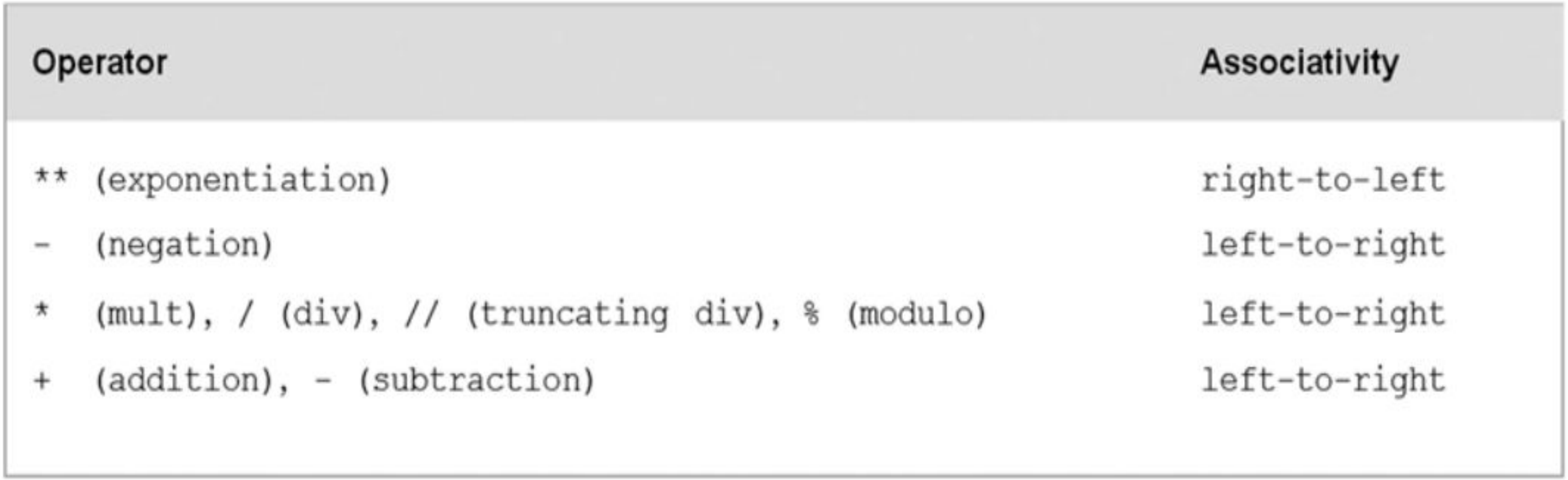
```python
k=10
print(k)
print(k*2)
print(k+2*k)
print((k+2)*k)
print(k*2/5)
print(k*(2/5))
print((k*2)/5)
print(k*2/5+8)
print(k*2/(5+8))
```

```
10
20
30
120
4.0
4.0
4.0
12.0
1.5384615384615385
```

pCPS 2.4.2 Operator Precedence in python

- The way one commonly represent **expressions**, in which **operators** appear **between** their **operands**, is referred to as **infix** **notation 10+20**

- There are other ways of representing expressions called **prefix** and **postfix** notation, in which **operators** are **placed** **before +10 20** and **after 10 20+** their **operands**, respectively



Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

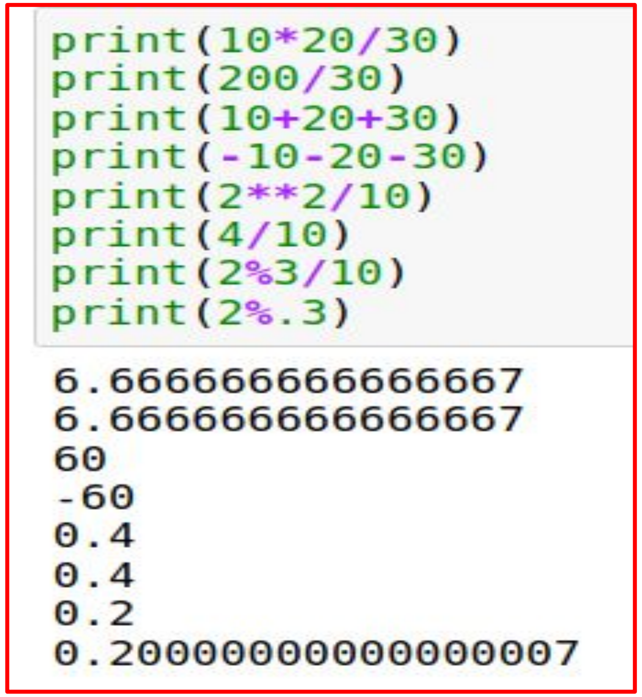
pCPS 2.4.2 Operator Precedence in python

- In the table, higher-priority operators are placed above lower-priority ones.

- Multiplication is performed before addition when no parentheses are included

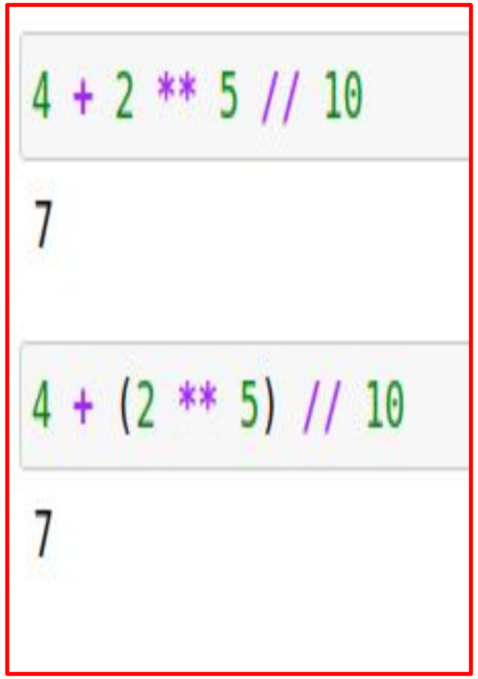
- Operator precedence guarantees a consistent interpretation of expressions.

- However, it is good programming practice to use parentheses even when not needed if it adds clarity and enhances readability, without overdoing it.



```
print(10*20/30)
print(200/30)
print(10+20+30)
print(-10-20-30)
print(2**2/10)
print(4/10)
print(2%3/10)
print(2%.3)
```

```
6.666666666666667
6.666666666666667
60
-60
0.4
0.4
0.2
0.20000000000000007
```



```
4 + 2 ** 5 // 10
```

```
7
```

```
4 + (2 ** 5) // 10
```

```
7
```

Following Python’s rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator

```
4 + 2 ** 5 // 10 → 4 + 32 // 10 → 4 + 3 → 7
```

The above expression would be better written as

```
4 + (2 ** 5) // 10
```

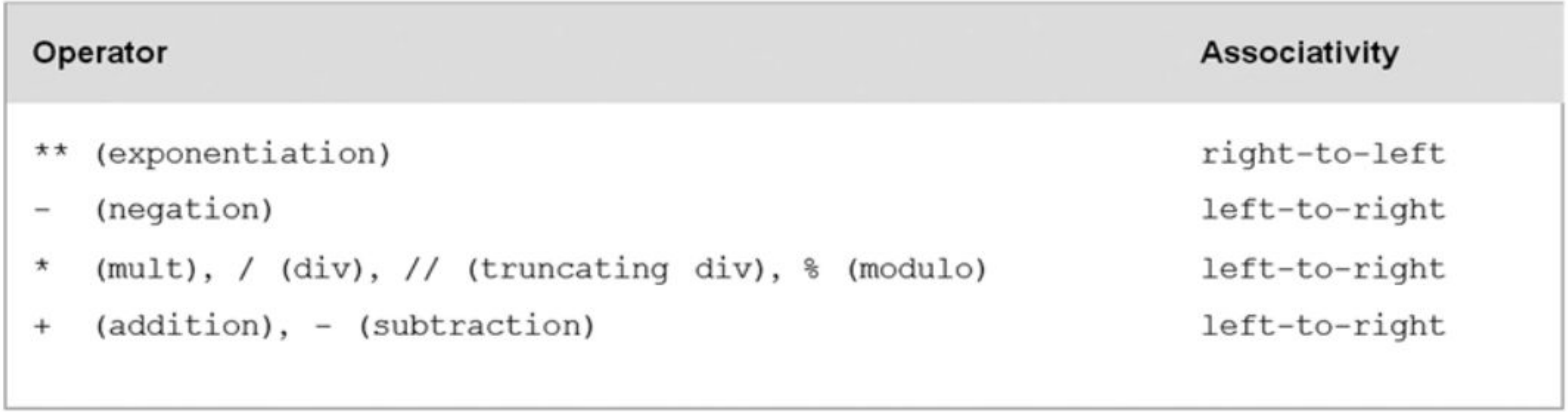
pCPS 2.4.3 Operator Associativity in python

- For operators following the **associative law**, the **order** of **evaluation doesn’t** matter

- Division and subtraction, do not follow the **associative law**

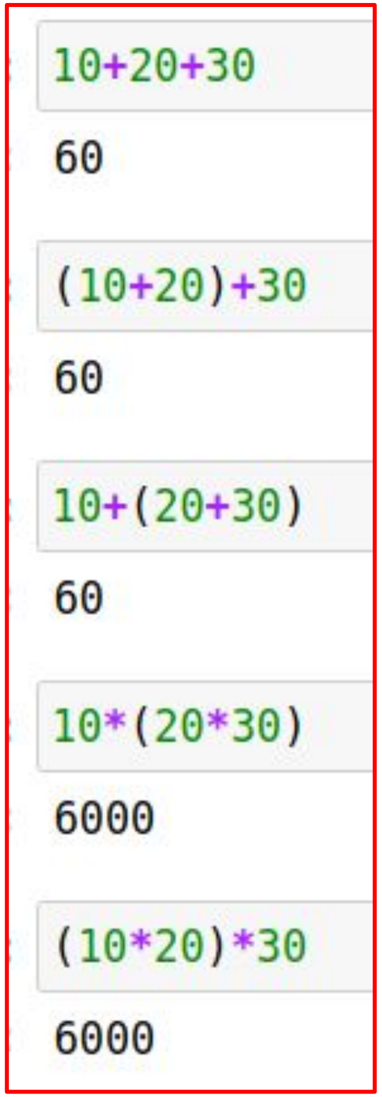
- Here, the order of evaluation does matter.

- To **resolve** the **ambiguity**, each operator has a **specified** operator **associativity** that defines the order that it and other operators with the **same level** of **precedence** are applied

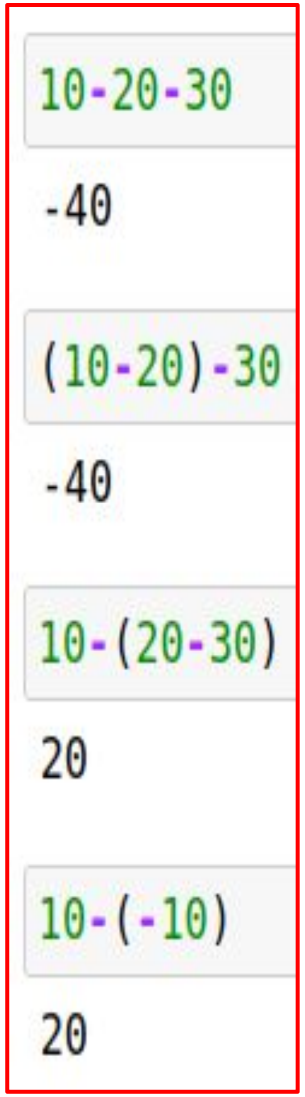


Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right

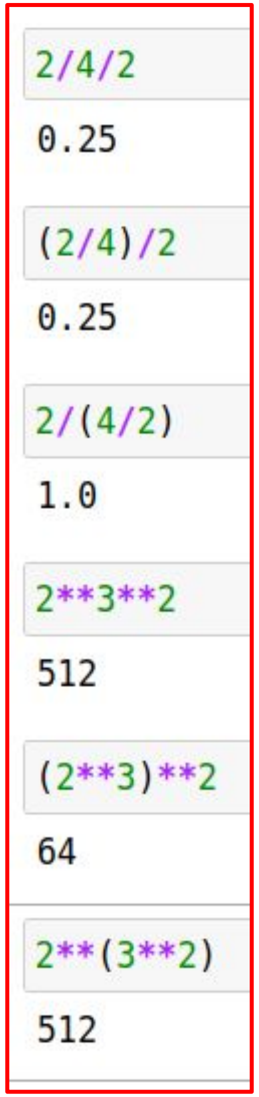
pCPS 2.4.3 Operator Associativity in python



```
10+20+30
60

(10+20)+30
60

10+(20+30)
60

10*(20*30)
6000

(10*20)*30
6000
```



```
10-20-30
-40

(10-20)-30
-40

10-(20-30)
20

10-(-10)
20
```



```
2/4/2
0.25

(2/4)/2
0.25

2/(4/2)
1.0

2**3**2
512

(2**3)**2
64

2**(3**2)
512
```

pCPS 2.4.4 Data Types in python

- A **data type** is a set of values, and a set of operators that may be applied to those values.

- For example, the integer data type consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among other operators.

- **Integers**, **floats**, and **strings** are part of a set of predefined data types in python called the built-in types .

- **Data types** prevent the programmer from using values inappropriately

- The need for **data types** results from the fact that the same internal representation of data can be interpreted in various ways

- If a programming language did not keep track of the intended type of each value, then the programmer would have to.

- This would likely lead to undetected programming errors, and would provide even more work for the programmer.

pCPS 2.4.4 Data Types in python

- There are two approaches to **data typing** in programming languages

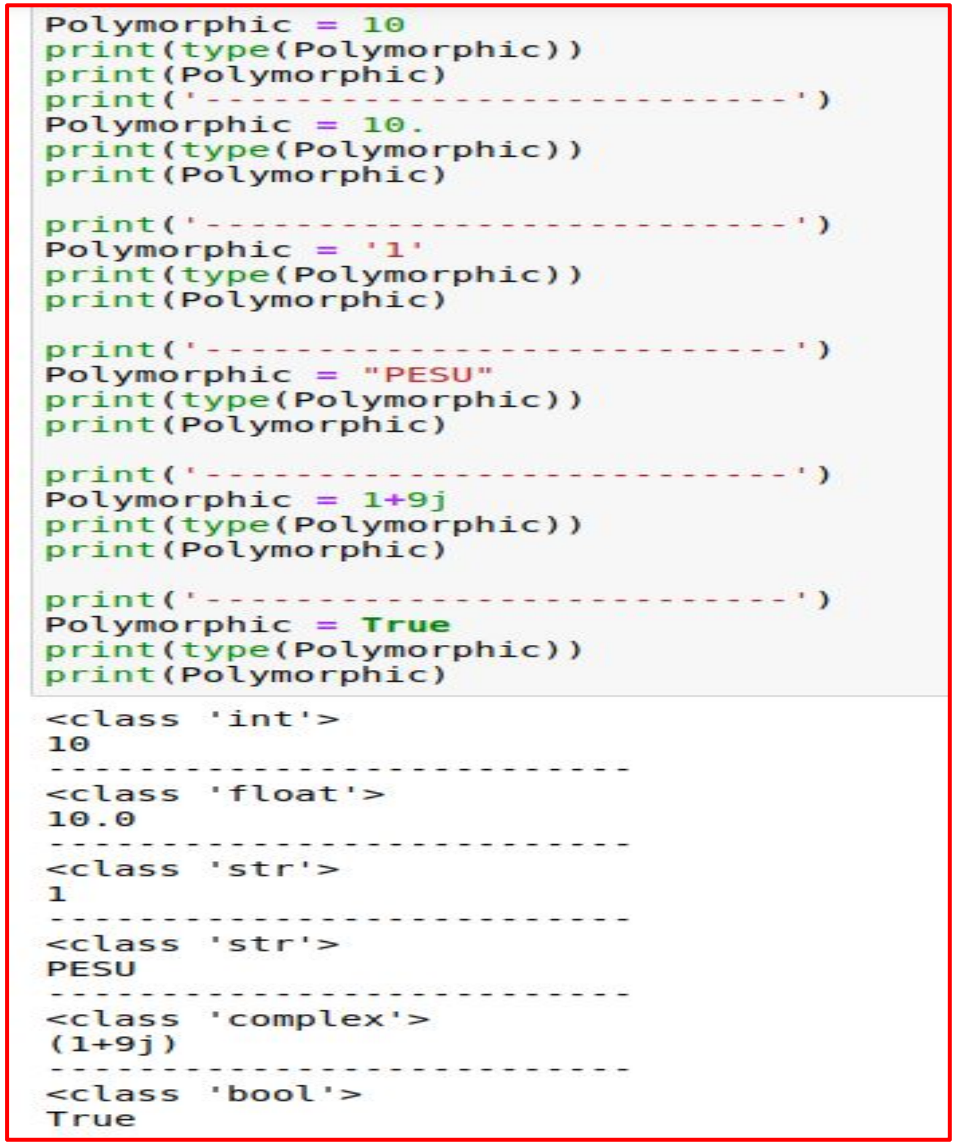
- In **static typing** , a variable is declared as a certain type before it is used, and can only be assigned values of that type.

- In **dynamic typing** , the data type of a variable **depends** only on the **type** of **value** that the variable is **currently holding**.

- The same variable may be assigned values of different type during the execution of a program

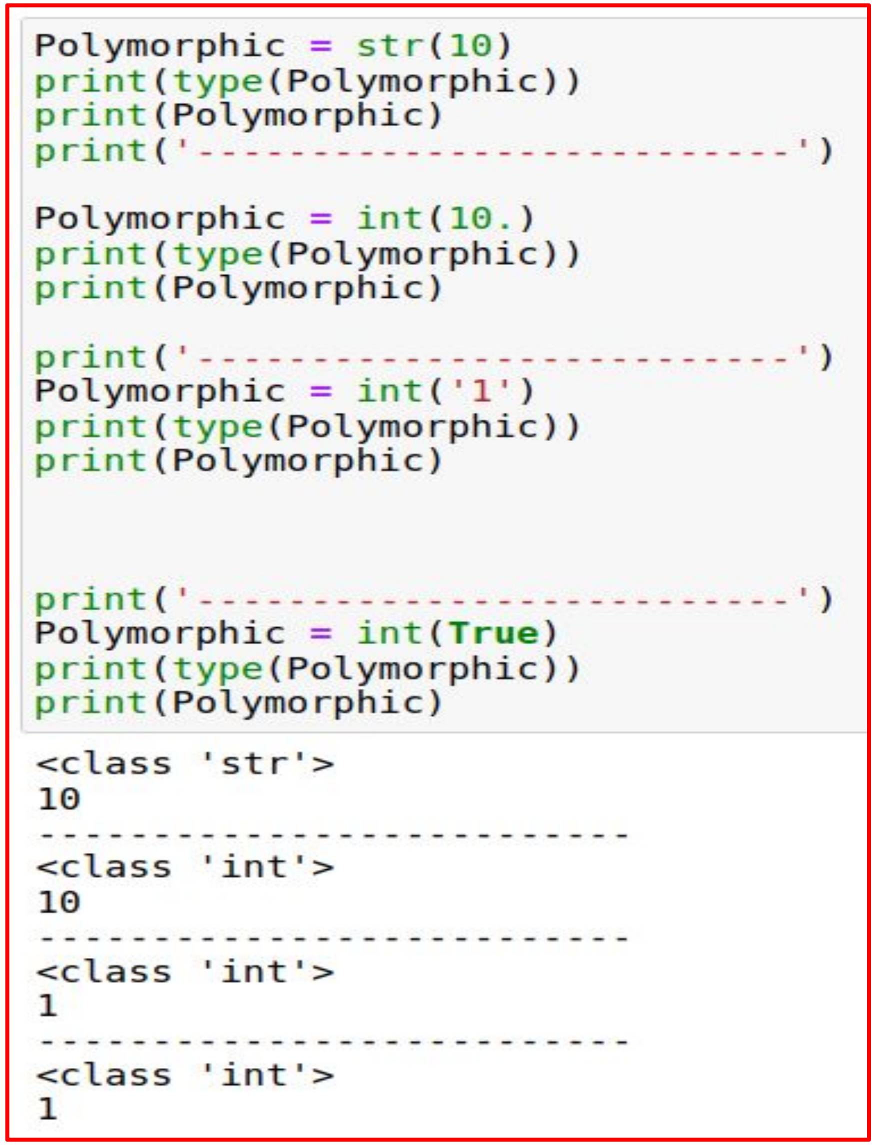
- **python**, uses **dynamic typing**

pCPS 2.4.4 Data Types in python



```
Polymorphic = 10
print(type(Polymorphic))
print(Polymorphic)
print('-------------------------')
Polymorphic = 10.
print(type(Polymorphic))
print(Polymorphic)

print('-------------------------')
Polymorphic = '1'
print(type(Polymorphic))
print(Polymorphic)

print('-------------------------')
Polymorphic = "PESU"
print(type(Polymorphic))
print(Polymorphic)

print('-------------------------')
Polymorphic = 1+9j
print(type(Polymorphic))
print(Polymorphic)

print('-------------------------')
Polymorphic = True
print(type(Polymorphic))
print(Polymorphic)
```

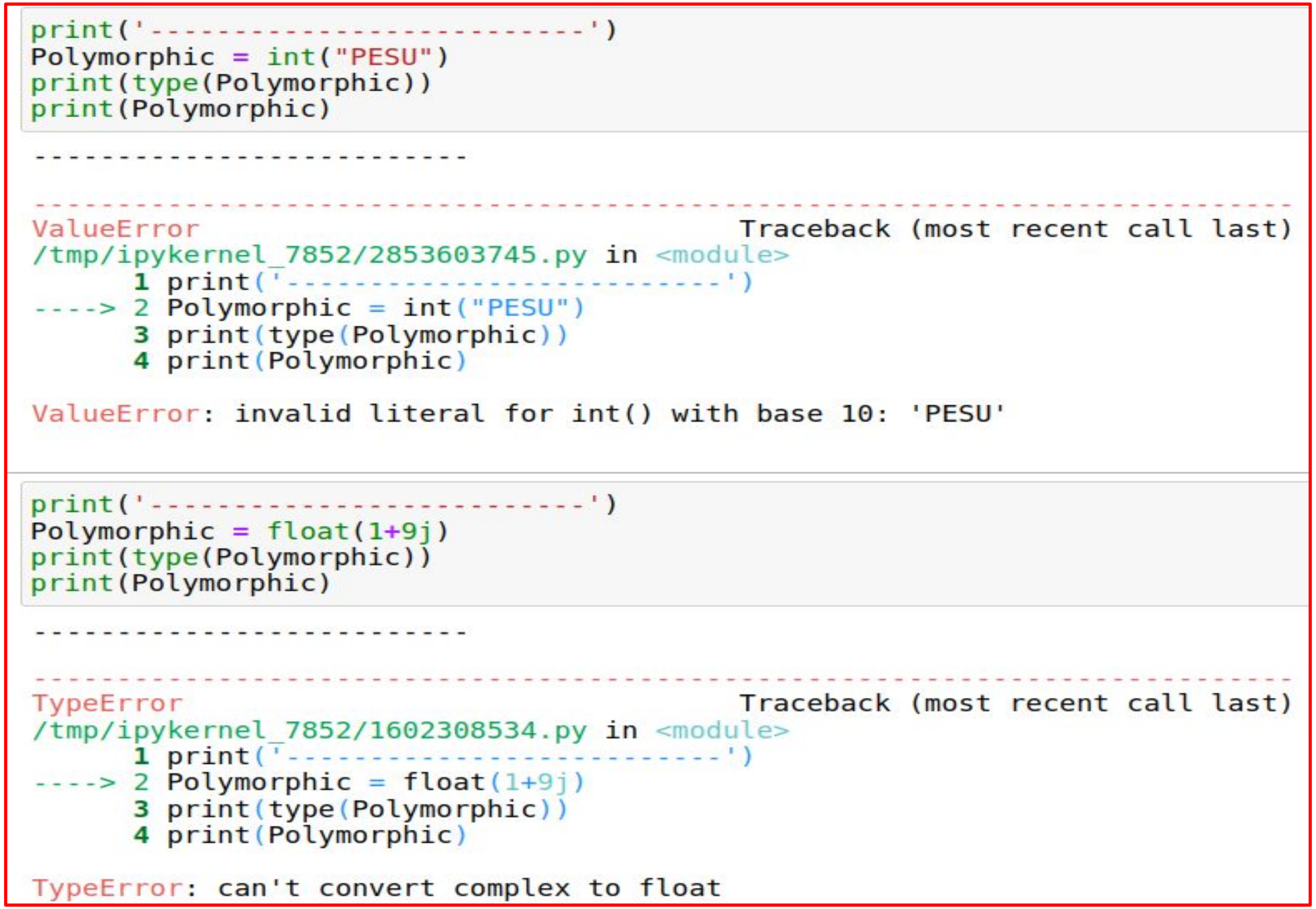
```
<class 'int'>
10
-------------------------
<class 'float'>
10.0
-------------------------
<class 'str'>
1
-------------------------
<class 'str'>
PESU
-------------------------
<class 'complex'>
(1+9j)
-------------------------
<class 'bool'>
True
```



```
Polymorphic = str(10)
print(type(Polymorphic))
print(Polymorphic)
print('-------------------------')

Polymorphic = int(10.)
print(type(Polymorphic))
print(Polymorphic)

print('-------------------------')
Polymorphic = int('1')
print(type(Polymorphic))
print(Polymorphic)



print('-------------------------')
Polymorphic = int(True)
print(type(Polymorphic))
print(Polymorphic)
```

```
<class 'str'>
10
-------------------------
<class 'int'>
10
-------------------------
<class 'int'>
1
-------------------------
<class 'int'>
1
```

pCPS 2.4.4 Data Types in python



```
print('--------------------------')
Polymorphic = int("PESU")
print(type(Polymorphic))
print(Polymorphic)
```

```
--------------------------

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_7852/2853603745.py in <module>
      1 print('--------------------------')
----> 2 Polymorphic = int("PESU")
      3 print(type(Polymorphic))
      4 print(Polymorphic)

ValueError: invalid literal for int() with base 10: 'PESU'
```

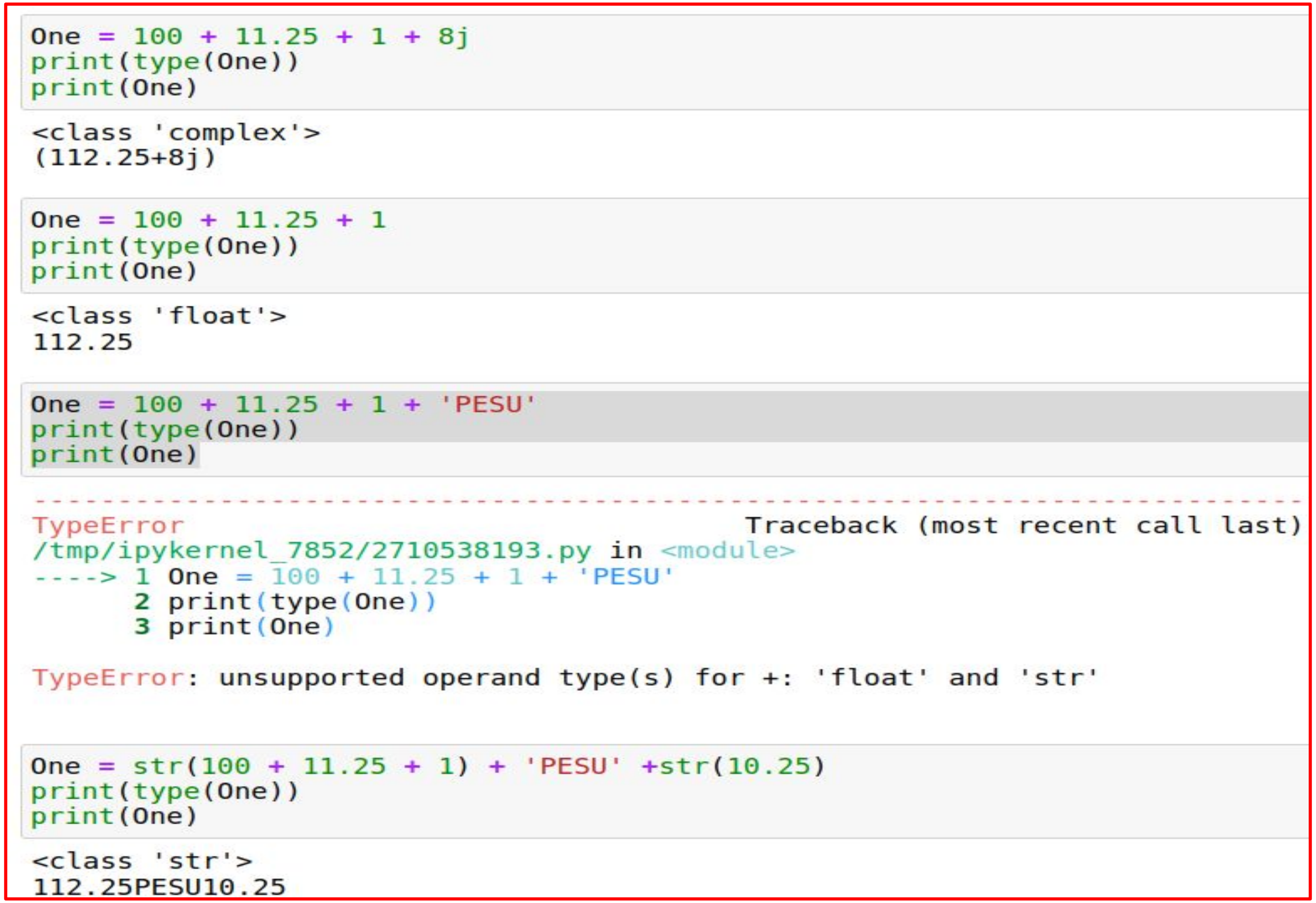
```
print('--------------------------')
Polymorphic = float(1+9j)
print(type(Polymorphic))
print(Polymorphic)
```

```
--------------------------

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_7852/1602308534.py in <module>
      1 print('--------------------------')
----> 2 Polymorphic = float(1+9j)
      3 print(type(Polymorphic))
      4 print(Polymorphic)

TypeError: can't convert complex to float
```

pCPS 2.4.5 Mixed type Expressions in python

- A **mixed-type expression** is an expression containing **operands** of **different** **type**.
- The **CPU** can only perform operations on values with the **same** internal representation **scheme**, and thus **only** on **operands** of the **same** type.
- **Operands** of **mixed**-type expressions therefore **must** be **converted** to a **common** type.
- Values can be **converted** in one of two ways namely by **implicit** (automatic) **conversion**, called **coercion** , or by **explicit** **type** **conversion**
- **Coercion** is the **implicit** (automatic) **conversion** of operands to a **common** type.
- **Coercion** is **automatically** performed on **mixed-type** expressions only if the operands can be **safely** converted, that is, if **no loss** of information will **result**.

pCPS 2.4.5 Mixed type Expressions in python



```
One = 100 + 11.25 + 1 + 8j
print(type(One))
print(One)
```

```
<class 'complex'>
(112.25+8j)
```

```
One = 100 + 11.25 + 1
print(type(One))
print(One)
```

```
<class 'float'>
112.25
```

```
One = 100 + 11.25 + 1 + 'PESU'
print(type(One))
print(One)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_7852/2710538193.py in <module>
----> 1 One = 100 + 11.25 + 1 + 'PESU'
      2 print(type(One))
      3 print(One)

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

```
One = str(100 + 11.25 + 1) + 'PESU' +str(10.25)
print(type(One))
print(One)
```

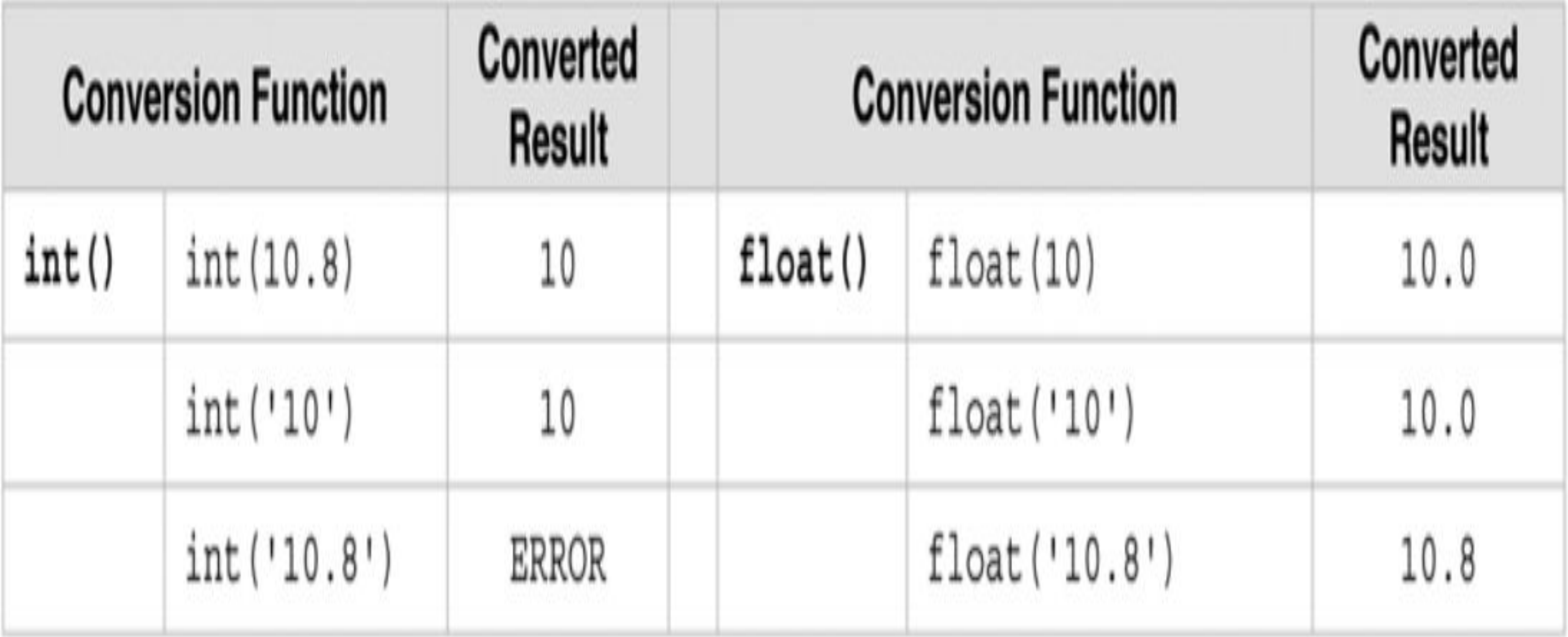
```
<class 'str'>
112.25PESU10.25
```

pCPS 2.4.5 Mixed type Expressions in python

- **Type conversion** is the explicit conversion of operands to a specific type.

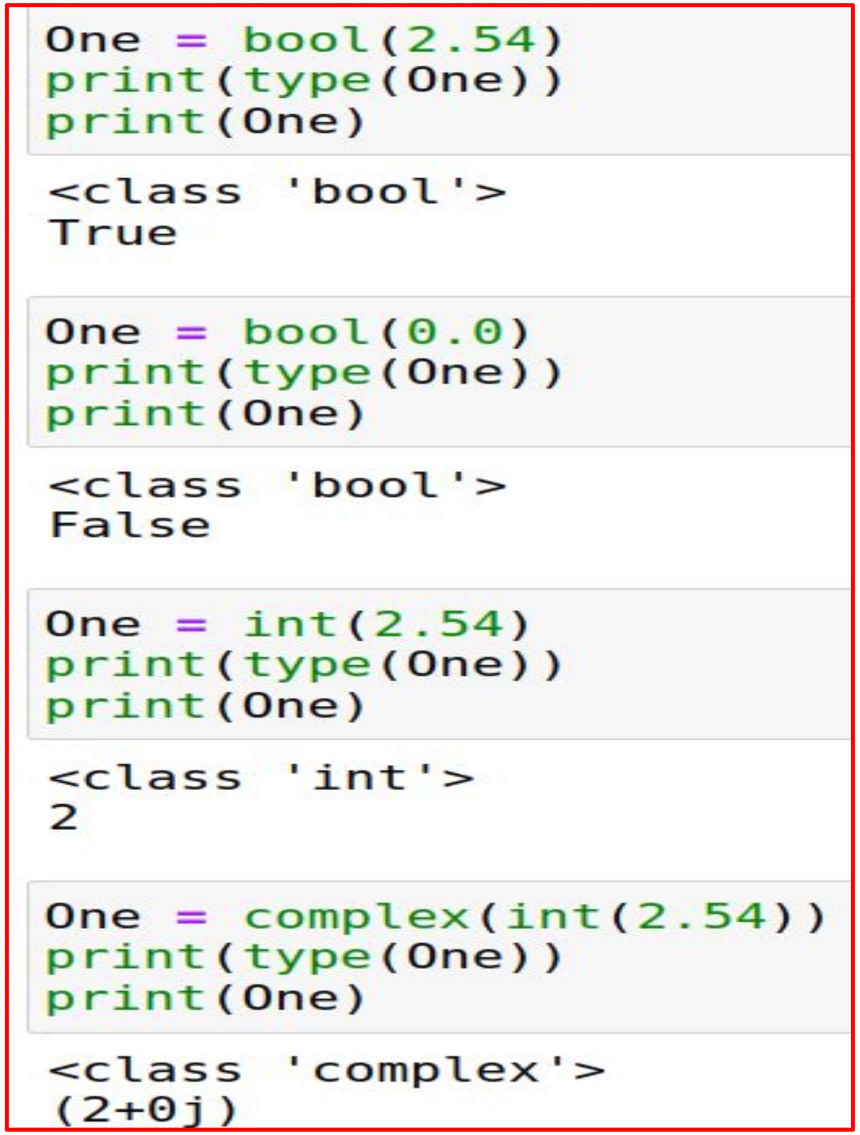
- **Type conversion** can be applied **even if loss** of information **results**

pCPS 2.4.5 Mixed type Expressions in python



Conversion Function		Converted Result		Conversion Function		Converted Result
int()	int(10.8)	10		float()	float(10)	10.0
	int('10')	10			float('10')	10.0
	int('10.8')	ERROR			float('10.8')	10.8

pCPS 2.4.5 Mixed type Expressions in python



```
One = bool(2.54)
print(type(One))
print(One)
```

```
<class 'bool'>
True
```

```
One = bool(0.0)
print(type(One))
print(One)
```

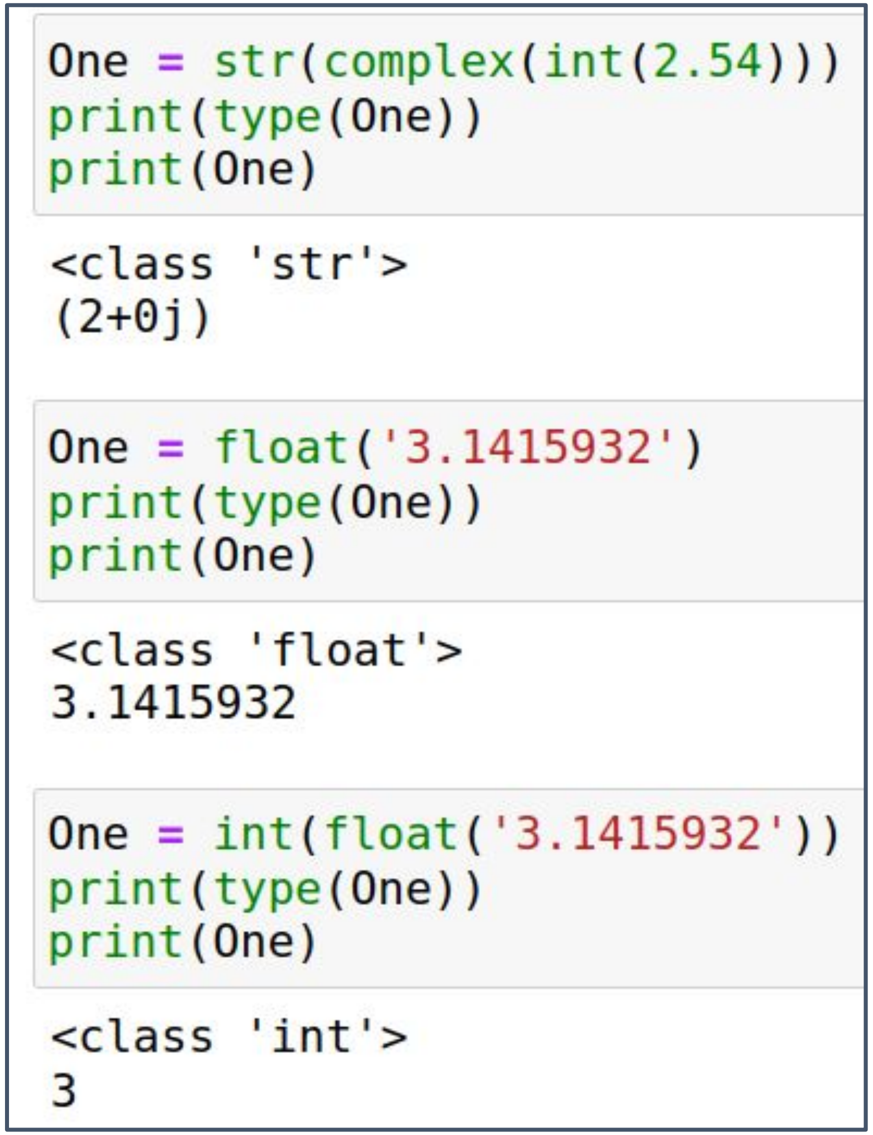
```
<class 'bool'>
False
```

```
One = int(2.54)
print(type(One))
print(One)
```

```
<class 'int'>
2
```

```
One = complex(int(2.54))
print(type(One))
print(One)
```

```
<class 'complex'>
(2+0j)
```



```
One = str(complex(int(2.54)))
print(type(One))
print(One)
```

```
<class 'str'>
(2+0j)
```

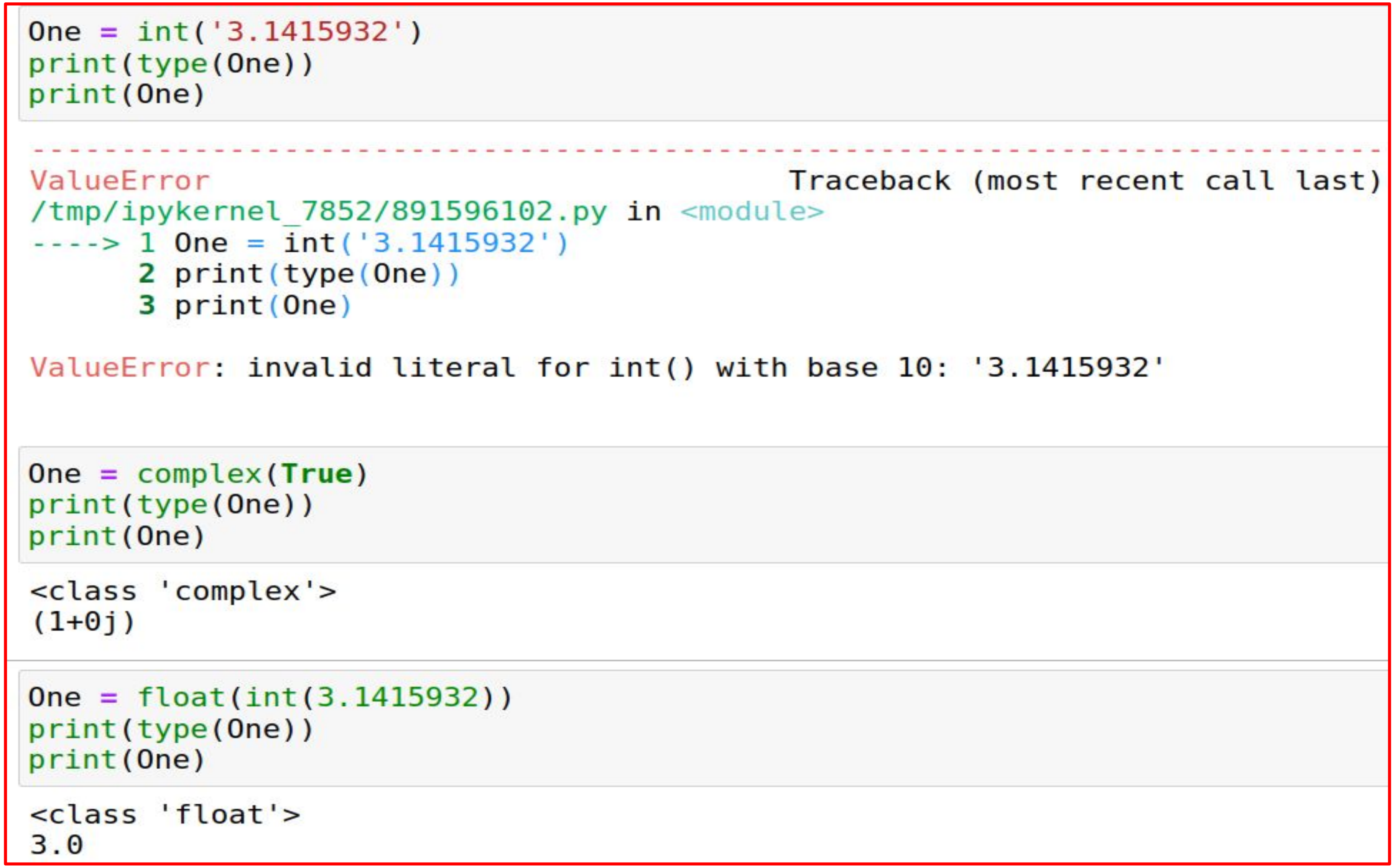
```
One = float('3.1415932')
print(type(One))
print(One)
```

```
<class 'float'>
3.1415932
```

```
One = int(float('3.1415932'))
print(type(One))
print(One)
```

```
<class 'int'>
3
```

pCPS 2.4.5 Mixed type Expressions in python



```
One = int('3.1415932')
print(type(One))
print(One)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_7852/891596102.py in <module>
----> 1 One = int('3.1415932')
      2 print(type(One))
      3 print(One)

ValueError: invalid literal for int() with base 10: '3.1415932'
```

```
One = complex(True)
print(type(One))
print(One)
```

```
<class 'complex'>
(1+0j)
```

```
One = float(int(3.1415932))
print(type(One))
print(One)
```

```
<class 'float'>
3.0
```

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University
nitin.pujari@pes.edu

For Course Digital Deliverables visit www.pesuacademy.com